UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/712,463	11/12/2003	Judith Schwabe	P-4181CIP	9131

24209 7590 04/01/2009

GUNNISON MCKAY & HODGSON, LLP
1900 GARDEN ROAD
SUITE 220
MONTEREY, CA 93940

EXAMINER	
VU, TUAN A	

ART UNIT	PAPER NUMBER
2193	

MAIL DATE	DELIVERY MODE
04/01/2009	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

UNITED STATES PATENT AND TRADEMARK OFFICE

Commissioner for Patents
United States Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450
www.uspto.gov

BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

Application Number: 10/712,463
Filing Date: November 12, 2003
Appellant(s): SCHWABE ET AL.

Forrest Gunnison
For Appellant

EXAMINER'S ANSWER

This is in response to the appeal brief filed 12/08/09 appealing from the Office action mailed 3/27/08.

(1) Real Party in Interest

A statement identifying by name the real party in interest is contained in the brief.

(2) Related Appeals and Interferences

The examiner is not aware of any related appeals, interferences, or judicial proceedings which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

(3) Status of Claims

The statement of the status of claims contained in the brief is correct.

(4) Status of Amendments After Final

The appellant's statement of the status of amendments after final rejection contained in the brief is correct.

(5) Summary of Claimed Subject Matter

The summary of claimed subject matter contained in the brief is partially correct in view of some Examiner's observations made in the Response to Arguments, section (10).

(6) Grounds of Rejection to be Reviewed on Appeal

The appellant's statement of the grounds of rejection to be reviewed on appeal is correct.

(7) Claims Appendix

The copy of the appealed claims contained in the Appendix to the brief is correct.

5,740,441	Yellin et al	4-1998
6,308,317	Wilkinson et al	10-2001

(9) Grounds of Rejection

The following ground(s) of rejection are applicable to the appealed claims:

/*******/

Claim Rejections - 35 USC § 103

1. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

2. Claims 1-78 are rejected under 35 U.S.C. 103(a) as being unpatentable over Yellin et al., USPN: 5740441 (hereinafter Yellin), and further in view of Wilkinson et al., USPN: 6,308,317 (hereinafter Wilkinson).

As per claim 1, Yellin discloses a method for arithmetic expression (e.g. col. 4, lines 42-51; *integer add* - col. 9, lines 42-52; col. 17, TABLE 1: *isub, idiv, imul, iadd*) optimization, comprising:

validating at least one input stack associated with a first instruction configured to operate on at least one operand of a first type, each of said at least one input stack associated with an input instruction of said first instruction, each input stack representing the state of an operand stack associated with an input instruction upon execution of said input instruction (e.g. Fig. 4B-D; col. 6, lines 6-38);

reconfigure said first instruction to operate as one operand of a second type, said second type smaller than said first type (e.g. *overflow* – Fig. 4C, 4D; *updated...modified ... by the*

current instructions - col. 10, line 58 to col. 11, line 6; *and improving execution time efficiency* – col. 5, line 65 to col. 6, line 4), said reconfiguring based at least in part on the relative size of said first type and said second type (Note: overflow analysis for each successor instruction reads on size of destination place in stack for a successor operand being smaller to take required size of upper instruction in the data flow – see Fig. 4A-B; *different data types* -- col. 21, lines 16-38); and

matching said second type with an operand type of at least one operand in said at least one input stack associated with said second instruction, said matching comprising a chain of instructions (e.g. Fig. 4, 5 – Note: any process that iterates one instruction at a time until all are processed reads on chain of instructions bounded by a start and a current element being processed), said chain bounded by said second instruction and a third instruction that is the source of said at least one operand (see col. 19-21: Do until there are no instructions whose changed bit is set … Do for each successor instruction …Merge the Virtual stack … Verification Success – Note: algorithm to merge – see col. 11-12).

But Yellin does not explicitly that (i) the above reconfiguring includes *converting said first instruction to a second instruction different from the first instruction* and operable *with a one operand of a smaller type than said first type*; nor does Yellin explicitly disclose that (ii) the above matching within a bounded chain includes changing the type of instructions in a chain of instructions to equal said second type *if said operand type is less than said second type.*

As for (ii), Yellin's matching (see col. 19-21: Do until there are no instructions whose changed bit is set … Do for each successor instruction …Merge the Virtual stack … Verification Success – Note: algorithm to merge – see col. 11-12) amounts to a flow wherein early elements

are matched against a requirement; and in a similar approach to boost execution of bytecodes, Wilkinson discloses iterative passes to match received bytecodes against the required stack size needed to execute the original code at the intended platform (Fig. 5-6) using a verifier and structures thereby to reconfigure length for representing a type of a loaded class or methods (col 14 line 23 to col. 15 line 17). One of ordinary skill in the art would recognize Yellin's iterative verification analogous to a matching so that for each candidate operand within a chain of instructions (see Wilkinson: *byte codes 60* Fig. 6) that is bounded between the source of the operand (*a third instruction* in the upper stream) where said target destination like a space of target *stack address* would be determined if it has to be modified to suit the required smaller size operand (or *said second type* operand of *the second instruction*); and this scenario corresponds to a chain in which source candidate operand type, earlier in the chain, is being loaded based on its loading order (from low to high) being *less than* the target stack address for the second type operand of *second instruction* (*); a fetching order which is equivalent on the 'being less than said second type' limitation.

As for (i), Yellin discloses adding data and rearranging stack calling structure relative to discrepancies of first and second type (see col. 1, lines 60-67; col. 20, lines 24-35; col. 21, lines 12-37; step 394, Fig. 4B) hence has suggested modifying the runtime environment as the loaded instructions are being verified in conjunction with stack settings to accommodate data being received from, for example, a larger size operand context on to a smaller size operand context (see col. 5, lines 14-64; *multiple computer platforms, underlying instruction sets* - col. 1, lines 7-14) hence the concept of accommodating runtime environment based on the nature of the received instruction to match platform, architectural, hardware constraint via reducing structures

needed to operate the original instruction or expanding the needed structures is recognized (see overflows, underflows – col. 15). Analogous to Yellin's approach as to modifying stack runtime Java bytecodes in order to accommodate an executing platform receiving/using data coming from a platform with higher architecture base (see Yellin: Background of the invention, col. 1), Wilkinson discloses using multiple passes (Fig. 6) whereby the received original bytecodes undergo successive modifications wherein stack operands are changed from a larger base platform to operands of lesser size that would fit the target platform (see Fig. 10-11; col. 11, lines 4-48), hence discloses a form of conversion from a word operand to a byte size operand - a lesser size than the word-based type operand.

As for (i) and (ii) based on the common endeavor by Yellin or Wilkinson to alleviate extraneous security and runtime resources of a given lower platform (e.g. receiving a JVM application destined for larger microcomputer into integrated circuit's microcontroller in portable devices) when loading operands for each instruction in Java method prior to runtime, it would have been obvious for one skill in the art at the time the invention was made to provide the operand type replacement as approached by Wilkinson so to support Yellin's loading process by way of converting at the receiving platform a larger type operand to a smaller size operand, and provide matching to see if candidates from a chain as taught in Wilkinson's multi-pass algorithm would be able to be subjected for such replacement, i.e. the matching where sequence of operands are brought in order to match (see *) a required size to accommodate for the intended platform architecture. One would be motivated to do so because providing operands of smaller size and familiar to the (smaller target platform) runtime as replacement for those corresponding operands of a higher size base as purported by Wilkinson and via Yellin's attempt to obviate

overflow would alleviate Yellin's application resources for dealing with exception due to overflow, obviate the resources of the JVM interpreter role as desired by Yellin (see by Yellin: col. 15, lines 12-30), and according to Wilkinson, the stack space of the smaller platform (e.g. Fig. 21-24) might be loaded from a chain as set forth above (see *) with data compliant to the platform thus expediting code execution via prechecking and reshuffling of bytecode, operands (see Fig. 9-11), enhancing the runtime with a safer method supporting a device whose resources are not equipped for checking large data being provided (see Wilkinson: BACKGROUND, col 3).

As per claims 2-3, see Yellin as said first instruction is arithmetic (col. 4, lines 42-51; *integer add* - col. 9, lines 42-52; col. 17, TABLE 1: *isub, idiv, imul, iadd*); a non-arithmetic, type-sensitive instruction (see col. 16-18: Table 1).

As per claims 4-5, see Yellin (in view of Wilkinson's conversion) for repeating said validating, said converting and said matching for instructions that comprise a program (see Figs 4-5) and linking each instruction to input instructions in all control paths (step 366 – Fig. 4A).

As per claim 6, see Yellin (col. 5, lines 14-64) wherein said first instruction is defined for a first processor having a first base; and said second instruction is defined for a second processor having a second base.

As per claims 7-8, Yellin wherein said first processor comprises a Java Virtual Machine (see Fig. 1-2); and in view of the obviousness rationale of claim 1 see Wilkinson (e.g. Fig. 1-2) wherein said second processor comprises a Java Card Virtual Machine with resource-restraint platform, i.e. operable with lower operand type; according to which rationale, Yellin combined

with Wilkinson(see Wilkinson Fig. 11) discloses wherein said first processor comprises a 32-bit processor; and said second processor comprises a resource-constrained 16-bit processor.

As per claims 9-10, Yellin discloses said at least one input stack comprises a plurality of input stacks, said plurality of input stacks further comprising a first input stack and a second input stack; and said validating further comprises comparing operand types of corresponding entries in said first input stack and said second input stack (refer to claim 1 in light of Figs. 4); wherein said comparing further comprises indicating an error if the types of at least said first at least one stack entry and said second at least one stack entry are not equivalent (Fig. 4D, 4E, 4F).

As per claim 11, Yellin by virtue of the overflow analysis and size mismatch (refer to claim 1) and the conversion from a larger size to a smaller size in claim 1, would have rendered (based on the teaching of Wilkison) obvious the following:

setting said second type to a smallest type;

setting said second type to the type of an operand in said at least one input stack if said smallest type is less than said type of said operand (see rationale of obviousness using teaching of Wilkinson); and setting said second type to a type that is larger than said smallest type if said smallest type is greater than said type of said operand, if said operand has potential overflow, if said second instruction is sensitive to overflow and if said second type is less than said first type (see rationale of claim 1; according to which any type if predicted to overflow beyond a smaller size will be replaced, i.e. overflow analysis as by Yellin inherently teaching setting a smallest type to a larger type when said smallest type is itself greater than type of any operand potentially risking being overflowed).

As per claim 12, Yellin (in view of Wilkinson) discloses wherein said smallest type is the smallest type supported by a target processor (see rationale of claims 7-8).

As per claims 13-14, Yellin (in view of Wilkinson) discloses wherein said smallest type is the smallest type determined during a previous pass of said converting (Yellin: Figs 4-5);

wherein said third instruction is not a source instruction (any instruction being loaded does not have to be a source for a operand that is predicted to overflow); and

said changing further comprises: recursively examining input instructions until said third instruction is obtained; and setting the type of said third instruction to equal said second type (Note: in view of the recursive process to verify by Yellin and changing to a smaller operand size of the target platform by Wilkinson, any replacement to match the platform operand size being included in the verification process – see Yellin: Fig. 4-5 – will read no setting a third instruction to be equal to a second instruction previously replaced in the chain)

As per claim 15, Yellin (in view of Wilkinson) discloses (see Verification algorithm: col. 19-21) wherein said third instruction comprises a source instruction (Note: within a chain of instructions source instruction is the identified instruction starting from which replacing a original first operand with target platform operand of a lesser size begins); and said

changing further comprises: recursively examining input instructions until said third instruction is obtained; setting the type of said third instruction to equal said second type; and repeating said changing for each input instruction of said third instruction (see rationale of claim 14).

As per claim 16, Yellin discloses comprising recording, said recording comprising: determining potential overflow associated with said second instruction (see col. 9, lines 30-60);

but Yellin does not disclose generating an output stack based at least in part on execution of said second instruction. But Wilkinson discloses replacement of operands for a stack (see Fig. 11, 16, 18) to support Yellin's verification endeavor as set forth in claim 1; hence the stack being modified to include (as an output stack) adjusted operands as per the combination set forth in claim 1 would also be an obvious limitation to enable Yellin to provide a smaller platform to make efficient use of its runtime environment, utilizing Wilkinson's approach for the reasons as set forth above in claim 1.

As per claim 17, Yellin discloses (combined with Wilkinson) wherein said determining further comprises: indicating said second instruction has potential overflow if said second type does not equal said first type, if said second instruction does not remove potential overflow, and if said second instruction creates potential overflow; and indicating said second instruction has potential overflow if said second type does not equal said first type, if said second instruction does not remove potential overflow, if said second instruction does not create potential overflow, if said second instruction propagates potential overflow, and if at least one operand in said at least one input stack has potential overflow (see Yellin: col. 9, lines 30 to col. 10, line 34 – Note: determining a need for change in a chain of instructions wherein operands of smaller size would be loaded in stack to *replace* – as set forth in claim 1 -- larger size operands deemed not suitable for the target platform reads on if said second instruction *does not remove* overflow or *might not necessarily create overflow*, and the automated process by which a chain in which all instructions are recursively verified reads on propagating the replacement due to potential overflow).

As per claim 18, Yellin discloses (combined with Wilkinson)wherein said determining further comprises: indicating said second instruction has potential overflow if said second type does not equal said first type, if said second instruction does not remove potential overflow, and if overflow is possible based at least in part on the type of said second instruction and the relationship between said first type and said second type; and indicating said second instruction has potential overflow if said second type does not equal said first type, if said second instruction does not remove potential overflow, if overflow is not possible based at least in part on the type of said second instruction and the relationship between said first type and said second type, if said second instruction propagates potential overflow, and if at least one operand in said at least one input stack has potential overflow (see Yellin in view of Wilkinson – as set forth in claim 1 -- according to the rationale of claim 17).

As per claim 19, Yellin alone does not disclose creating a new output stack based at least in part on one of said at least one input stack; updating said new output stack based at least in part on operation of said second instruction; and indicating another instruction conversion pass is required if said new stack does not equal a previous output stack. But in view of the automated and iterative verification process by Yellin combined with the operands replacement by Wilkinson as set forth in claim 1, the constant updating of stack per iteration pass in regard to what is deemed potentially conflicting in regard to stack overflow (see Yellin: Figs 4-5), it would have been obvious for one skill in the art to make many passes to update the runtime stack according to Yellin's approach using marking of unmatched operand allocation, so that combining with Wilkinson's replacement of larger size operands with more compliant size operands, Yellin's verification would be ready for execution with a stack that is repeatedly

updated, and this would further alleviate runtime resources of small platforms using smaller operand type as set forth in claim 1.

As per claim 20, Yellin discloses a method for arithmetic expression optimization, comprising: step for validating at least one input stack associated with a first instruction configured to operate on at least one operand of a first type, each of said at least one input stack associated with an input instruction of said first instruction, each input stack representing the state of an operand stack associated with an input instruction upon execution of said input instruction;

step for reconfiguring said first instruction to operate on at least one operand of a second type, said second type smaller than said first type, said converting based at least in part on the relative size of said first type and said second type; and

step for matching said second type with an operand type of at least one operand in said at least one input stack associated with said second instruction, said matching including a chain being bounded; said chain bounded by said second instruction and a third instruction that is the source of said at least one operand;

all of which limitations having been addressed in claim 1.

But Yellin does not explicitly that the above reconfiguring includes converting said first instruction to a second instruction different from the first instruction and operable with a one operand of a smaller type than said first type; nor does Yellin disclose that the above matching within a bounded chain includes changing the type of instructions in a chain of instructions to equal said second type if said operand type is less than said second type. Such limitation has been addressed in claim 1 using Wilkinson.

As per claims 21-38, refer to rejections set forth to claims 2-19 respectively.

As per claim 39, Yellin discloses a program storage device readable by a machine, embodying a program of instructions executable by the machine to perform a method for arithmetic expression optimization, the method comprising:

validating at least one input stack associated with a first instruction configured to operate on at least one operand of a first type, each of said at least one input stack associated with an input instruction of said first instruction, each input stack representing the state of an operand stack associated with an input instruction upon execution of said input instruction;

reconfiguring said first instruction to operate on at least one operand of a second type, said second type smaller than said first type, said converting based at least in part on the relative size of said first type and said second type; and

matching said second type with an operand type of at least one operand in said at least one input stack associated with said second instruction, said matching comprising a chain of instructions being bounded, said chain bounded by said second instruction and a third instruction that is the source of said at least one operand;

all of which limitations having been addressed in claim 1.

But Yellin does not explicitly that the above reconfiguring includes converting said first instruction to a second instruction different from the first instruction and operable with a one operand of a smaller type than said first type; nor does Yellin disclose that the above matching within a bounded chain includes changing the type of instructions in a chain of instructions to equal said second type if said operand type is less than said second type. But the limitation has been addressed in claim 1 using a combination with Wilkinson.

As per claims 40-57, refer to rejections set forth to claims 2-19 respectively.

As per claims 58-76, these are the apparatus version of claims 1-19; hence are rejected with the corresponding rejections as set forth therein, respectively.

As per claim 77, Yellin discloses a method of using an application software program including arithmetic expression optimization of at least one instruction targeted to a processor, the method comprising receiving the software program on said processor, said software program optimized according to a method comprising:

validating at least one input stack associated with a first instruction configured to operate on at least one operand of a first type, each of said at least one input stack associated with an input instruction of said first instruction, each input stack representing the state of an operand stack associated with an input instruction upon execution of said input instruction;

reconfiguring said first instruction to operate on at least one operand of a second type, said second type smaller than said first type, said converting based at least in part on the relative size of said first type and said second type; and

matching said second type with an operand type of at least one operand in said at least one input stack associated with said second instruction, said matching comprising a chain of instructions being bounded, said chain bounded by said second instruction and a third instruction that is the source of said at least one operand; and executing said at least one instruction on said processor;

all of which limitations having been addressed in claim 1.

But Yellin does not explicitly that the above reconfiguring includes converting said first instruction to a second instruction different from the first instruction and operable with a one

operand of a smaller type than said first type; nor does Yellin disclose that the above matching within a bounded chain includes changing the type of instructions in a chain of instructions to equal said second type if said operand type is less than said second type. But the limitation has been addressed in claim 1.

As per claim 78, Yellin discloses a controller configured to execute a virtual machine, the virtual machine capable of executing a software application comprising a plurality of previously optimized instructions, the instructions optimized by a method comprising:

validating at least one input stack associated with a first instruction configured to operate on at least one operand of a first type, each of said at least one input stack associated with an input instruction of said first instruction, each input stack representing the state of an operand stack associated with an input instruction upon execution of said input instruction;

reconfiguring said first instruction to operate on at least one operand of a second type, said second type smaller than said first type, said converting based at least in part on the relative size of said first type and said second type; and

matching said second type with an operand type of at least one operand in said at least one input stack associated with said second instruction, said matching comprising a chain of instructions being bounded, said chain bounded by said second instruction and a third instruction that is the source of said at least one operand;

all of which limitations having been addressed in claim 1.

But Yellin does not explicitly that the above reconfiguring includes converting said first instruction to a second instruction different from the first instruction and operable with a one operand of a smaller type than said first type; nor does Yellin disclose that the above matching

within a bounded chain includes changing the type of instructions in a chain of instructions to equal said second type if said operand type is less than said second type. But the limitation has been addressed in claim 1.

Nor does Yellin disclose that the controller is a smart card having a microcontroller embedded therein; but in view of the rationale as set forth in claim including Wilkinson's Java card machine, this smart card controller limitation would also have been obvious because of the benefits as set forth in that rationale.

/*****/

(10) Response to Argument

(A) Appellants have submitted that the words of the claim have not been considered and that the combination as proffered in the rejection teaches away from 'changing the type of instructions in a chain of instructions to equal said second type if said operand type is less than said second type' (claim 1) wherein 'said chain bounded by said second instruction and a third instruction that is the source of said at least one operand' (Appl. Rmrks pg. 19). The language of the whole claim has to be dissected and considered, including the *antecedent basis* relationship conveyed from the term like 'said second type', 'said second instruction'; and to this respect, one cannot dissociate the *matching* step from the *converting* step, both relating to a same *second instruction* operable with the same *second type operand*, as observed as follows.

From interpreting the claim language, 'said second type' is the smaller operand type associated with a second instruction, a type smaller relative to the first operand type associated with the first instruction, which is converted to the second instruction (see paragraph: *converting said first instruction ... is different from said first instruction*).

When operands in a chain of source instructions are matched with this 'second type' operand, the candidate source operands (as they are fetched in an order from lower to higher) in an item of this chain is deemed as being equal to the position of this 'second type' in the algorithmic flow or sequences of matching, the position taken by this 'second type' operand being the upper bound, and this is how 'matching' viewed as 'changing the type of instructions in a chain of instructions **to equal** said second type if said *operand type* is less than said second type' has been interpreted (emphasis added). When 'said second type' is understood as operand type, it is hard to interpret matching as actually changing instruction type to be equal to an operand type (i.e. a smaller size type), this very operand type operable with second instruction associated with operands in a input stack. One of ordinary skill in the art cannot see how a full instruction type which operates on operands can be made **equal to** an operand type (*said second type*), particularly when the claim does not make it clear how the input stack operands (or operand type) being matched against 'said second type' precisely correlate to the chain of *instruction type* involved in the matching and changing (if said *operand type* is less than said second type). This insufficient clarity in the claim is neither removed nor resolved when one consults the Appellants' Summary of Claimed Subject Matter in section 5 above (Brief: pg. 7, top).

(B) Independent claim 1: the language of 'matching *said second type* with an operand type of at least one operand in ... input stack *associated with said second instruction*' implicates 'said second type' associated with the second instruction and that this 'second type' operand is smaller than a first operand type of the first instruction from the previous paragraph of claim 1 (i.e. *second instruction ... to operate on ... one operand of a second type, said second type smaller*

than said first type). The paragraph 0067, and step Fig. 20 cited to correspond the above *matching* describes how an operand type is compared against an *instruction type*, not operand of said associated second instruction with a second type and smaller operand. The language recited as 'matching comprising changing the type of instructions in a chain of instructions to equal said second type if said operand type is less than said second type' is mapped with paragraph 0068, and accordingly, para 0068 and Fig. 21 fail to actually define the matching as recited, in terms of exact matching the structures and functions to the element of the claim. Indeed, paragraph 0068 does not teach about a operand (*second type* operand) associated with a *second instruction* being smaller than a first operand type, which when matched with a operand (of a stack) *associated with said second instruction* would trigger changing an instruction type (in a bounded chain whose upper bound is *said second instruction* operable with *said second type*) to be equal to *said second type* operand, if said stack operand is less than said second type (i.e. second type operand construed as operable with said second instruction, and smaller than operand type of a first instruction. Nor does paragraph 0068 teach a upper bound (of a chain) such that this upper bound is the very second instruction operable with a second type operand, which is smaller to a first operand, as construed from claim 1. Figure 21 describes receiving an instruction type, determining is instruction is to generate an operand, and matching this received type for each operand of input stack, such that the received instruction type is generated to accommodate the larger size of instruction type, and this is not depicting the language of 'matching … changing the type of instructions … to equal said *second type* if *said operand type* is less than said second type', with *said second type* clearly depicted as being associated with second instruction. Matching input stack operand types associated with *said second instruction* with *said second*

type operand cannot be same as matching instruction type in a chain and equating this matching with replacing this *instruction type* to be equal to said *second type operand* if operand type (of the stack) is less than the latter, when the steps of Fig. 21 does not mention about stack operands pertinent to instruction in a chain being compared with 'second type operand' being smaller than first operand type. Nor does Fig. 21 and Fig. 20 relates said *second instruction with smaller operand type* replacing a first instruction, and how every instruction type in the chain bounded by this second instruction is rendered equal to just a mere operand type (said second type) in a process of matching one operand in a stack associated with this second instruction; when the claim requires that second type operand is matched against *operand in a stack associated with second instruction*, the second instruction operable with the second type having been recited as being smaller to a first type operand.

In short, the provided paragraphs do not distinguish two type of conversions stages cleanly separated in time, using the result (i.e. said second type being the smaller operand)of the first conversion (from larger to smaller – first instruction to said second instruction), and implement a second stage conversion from smaller to larger as argued; nor do the above cited paragraphs depict second instruction operable with a second operand type ('said second type') being *smaller than* a first operand type associated with the first instruction, which has been converted into the very second instruction, recited in the matching step as delimiting a chain of instructions which Appellants now proffered as a chain where all the matched instruction type therein are converted from a smaller 'type' to a larger 'type' being this 'said second type'. The *matching* step cannot rely on the Specifications to make any sense or to come to agreement with the 'conversion' (smaller to larger) as set forth in the Arguments.

(C) The matching limitation should be understood and has been interpreted as subsumed under (or being a complementary action within) the converting step (first instruction, associated with a first operand, to a second instruction operable with a second operand) recited in claim 1, wherein said second operand is a type smaller than the first type operand, and that the first instruction associated with the first type is converted to a second instruction operable with said second type operand, which is para 0065, pg. 35. Accordingly, the matching in terms of changing a instruction type to be equal to said second type (if said operand type is less than said second type) has been interpreted as set forth in (A). Referring to section (B), the *matching* limitation and *changing* in the context of 'less than said second type' has not been given weight (emphasis added) of a conversion of code instruction from one instruction type to another instruction type of a larger size operand, as stressed in the Appellants (Appl. Rmrks pg. 20-21). That is, a sequence of received instructions is matched against a target type based on stack size or architecture constraints, so that algorithmic steps enable all instructions to be matched (as being spatially and temporally equal for comparison against the required size needed to realize the instruction within the target platform) and this sequential approach is deemed fulfilling 'if said operand type is less than said second type'. *Operand type* here is referred to any operand which is associated with the second instruction as in the perspective of stack of the target platform; e.g. which can require downsizing or expanding to fit the platform, and 'said second type' is interpreted as operand type (e.g. smaller than a first operand type) standardized to suit the very target platform, such that all candidate instructions leading to this reference 'second type' would belong to the chain including all the pertinent operand type to match against the upper bound reference 'second type'. This interpretation has been set forth in the rejection of

claim 1 to address 'if said operand type is less than said second type' in the context of a bounded chain. The claim language (claim 1, 20) does not specify what exactly 'is less than' really amounts to, hence broad interpretation has been utilized; nor does the claim establish true relationship between 'operand type' (in input stack) with *instruction type* in a chain of instructions or that of third instruction being the source of one operand. One of ordinary has to apply broad interpretation because the Summary of the claimed subject matter fails to teach how instruction type in a bounded chain is made equal to a second type operand associated with a second instruction (or upper bound), when this second type is not depicted in the cited parts of the Summary as being smaller in size to a first operand type belonging to first instruction, the conversion of which is this second instruction being the upper bound of a chain. An instruction type operates in conjunction with objects known as operands, and modifying instruction so that it would properly function with the required size of the operands will be more credible scenario, which Wilkinson teaches in light on the platform size discrepancies addressed in Yellin. An instruction is composed on operator and operands, with instruction being realized from executing operators (pertinent to instruction type dictated by the architecture) upon the objects being the operands. But first claiming that matching a operand type (said *second type*) with another operand type so that, later, and instruction type in a chain is changed to be equal to this 'second type' would be deemed far-fetched, thereby triggering broad interpretation regarding the 'to be equal to' and 'if less than said second type'; all of which explained in the 103 Rejection.

(D) Yellin in view of Wilkinson in combination has rendered obvious converting original instructions so that based on platform architecture (i.e. with emphasis to overflow concerns) the original instruction are modified to fit stack operand requirement or needed to execute the

instructions when these are ported in a smaller size architecture. The rejection has interpreted the matching limitation and has given its weight only to the extent of matching in order to accommodate received bytecodes (as in Yellin or Wilkinson) to the size of the target platform in terms of mostly downsizing type modification of the original byte code instructions. As the Appellants globally assert that the matching is about converting from a smaller to a larger type (Appl. Rmrks pg. 21) the learning about this 'type' is not definite nor is it corroborated with facts, since the arguments fail to point out exactly *smaller type* being instruction or of operand type; nor larger type being operand or larger type of instruction. One would wonder whether the matching step is about converting among operands only; or from instruction into operand; or instruction into another instruction. As for the matching step, the claim recites 'said second type' and clearly makes this second type as a smaller operand type for which conversion is required. One cannot see how a smaller type operand (*said second type*)would be suddenly contemplated as a *larger operand* or instruction type (as repeatedly alleged in the Arguments (Appl. Rmrks pg. 20-21) that would suddenly require that all stack operands associated with this 'second instruction' (i.e. *instruction* operable with this second and smaller type operand – see claim 1) be viewed *as less than* 'said second type' whereby enabling changing a *instruction type* in a chain bounded by the second instruction so to make *instruction type* made equal to this *second type* operand, in light of the deficiency of the Summary of Claimed subject matter as raised in section (B). Since operand type is recited as from an input stack associated with the second instruction, the relationship of this stack operand and any instruction in the bounded chain is unclear, rendering 'said operand type is less' even less integral to any *instruction type* of the chain. The language recited as 'if said operand type is less than said second type' is not

elaborated sufficiently for one of ordinary skill to be convinced that the changing is really about making one such *instruction type* equal to this *second type* of operand (in claim 1 as a whole) or even, as alleged by the Appellants, which changes instruction type from a lower size *instruction type* to a bigger size *instruction type*. The argument that Yellin in combination with Wilkinson teaches away from 'converting a smaller type' to a 'larger type' is deemed not persuasive, because as a whole the scenario of claim 1 still conveys that 'said second type' is a smaller operand operable with said second instruction, and every converting there is should be understood in light of this required relationship. The paragraph 0067, 0068 of the Summary of subject matter do not mention about upper instruction bounding corresponding to a operand type which is smaller (than a first operand type) and that this second (smaller) operand type is associated with the second instruction at said upper bound. Further, changing instruction type in a chain bounded by this second instruction in terms of making instruction type equal to a smaller operand type associated with this upper bound second instruction is no where understood from the Summary of the claimed matter, nor can be accepted by one of ordinary skill in the art.

(E) The claimed scenario does not disassociate 'said second type' and 'said second instruction' from the *converting* from a larger operand type to a smaller operand type context, but in fact, intertwines 'second type' operand and 'second instruction' recited in the first converting step (from a larger operand to a smaller operand type instruction) with the *matching* step, which turns out to be what Appellants alleged as a converting from a "smaller type" to a "larger type", with 'type' mentioned in a indecisive manner (see Appl. Rmrks pg. 20-21), allegation hard to accept when the language of the claim teaches not converting but matching having a changing to equate *instruction type* with *said second* operand *type*, based on operand type of a input stack

(emphasis added). *Antecedent basis* regarding this 'said second type' cannot propel this 'said second type' to be a larger operand type, based on which the matching step would change any (questionably asserted) *smaller* instruction **type** within a chain to be EQUAL to such 'said second type', asserted by Appellant as a larger instruction type, not a (smaller) said *second* operand *type*. No part of the Disclosure teaches this far-fetched form of equating as alleged in the Arguments; therefore, one of ordinary skill in the art to employ broadest and reasonable interpretation to interpret the language 'changing the type of instructions to equal said second type'; that is, the algorithm by which candidate original instructions is matched against a reference operand type associated with a target second instruction (that would executes in the target platform) provides the spatial and temporal setting where successively, a original instruction is made equal to that of the contemplated target reference instruction, wherein the iteratively step of matching each instruction (each including a operand type) within a chain of such candidate instructions being bounded by the reference target second instruction reads on if said operand type less than said second type, since second type operand is inherent to said second instruction.

The argument that as combined Yellin and Wilkinson does not teach nor render obvious 'converting smaller instruction type to a larger instruction type' is deemed insufficient, because the language of the claim as a whole has been interpreted and addressed accordingly. The rationale of obviousness stands.

(F) Appellants have submitted that the Yellin reference has been mischaracterized and misused by the Office Action, especially regarding modifying of bytecodes for porting to a different platform (Appl. Rmrks, pg. 22-23) and that in modifying code as asserted by the Office

action, Yellin would have security breach (Appl. Rmrks pg. 24). It was a well-known concept that bytecodes used in Yellin's Java engine can serve to port program code created in a source operating system to another environment having an operating system different from the source operating environment, in view of the portability of this form of code. To this effect, Yellin discloses a verification process by which loaded bytecodes in distributed environment including different platforms (col. 5 lines 18-53) can handle discrepancies in terms of stack requirement – operand type compatibility - or stack overflow concerns, i.e. allowing a larger size instruction being loaded onto a platform whose stack cannot support that size. Yellin discloses modifier code effectuating on the stack data type to address the required bit changes or address space requirement when instruction are verified (see pseudocode - col. 20-21; Fig. 4A, 4B) by replacing data type for the stack or register based on ancestor handles and effectuate merging of stack values. The verifier approach in Yellin is purported to accelerate execution of the bytecodes with the certainty that overflow type issues regarding compatibility of data type is checked, and nothing in Yellin verifier code intervention and stack data replacement indicate any form of breach in security as alleged by the Appellants. In the same line of bytecode application across heterogeneous machines, Wilkinson teaches source bytecodes to be loaded to be larger in size than the target platform, with the particularity that in Wilkinson, size of target architecture (an integrated circuit environment) is such that the original bytecodes are to be converted to match its architectural stack in its version of JVM. The103 rejection is purported to render obvious the converting step (from a larger operand instruction to a smaller operand instruction) in terms as to how portable bytecodes, when received into a particular target context, could be subject to modification so that it would accommodate the architecture and stack limitation of

another platform (just as Yellin pursues), and in case that the target platform would be that of Wilkinson microcontroller OS, such form of conversion of instructions --as in Wilkinson, as these are loaded by the target runtime Java loader, would have been obvious in view of the concern to readjust prior to runtime set forth in Yellin to obviate exceptions and prevent overflows and underflows (see col. 15).

(G) Appellants have submitted that mischaracterization of Yellin is evident when the cited code by the verifier program is not concerned with rearranging and updating stack data structures as proffered by the Examination; but rather describes mere updating vector bits associated with subroutines, hence such vector updating by Yellin is not misunderstood as converting bytecode program (Appl. Rmrks pg. 26-27). Yellin's using the stack data replacement and updating acts by the verifier code has not been cited to anticipate 'converting' the received program (whose instructions are being fetched as they are loaded) from a larger operand original context to a smaller operand target context, as construed from the claim. This is a 103 Rationale, where teachings from Yellin are used to impart one solution to obviating memory conflicts (e.g. due to platform differentials, overflows) and improve runtime speed, via a form of manipulation or transformation by which platform compatibilities would be detected and addressed immediately prior to actual runtime of the received bytecodes. Yellin does provide sufficient pre-runtime manipulation by the target JVM environment by which received bytecodes into a target platform can be checked and manipulated to obviate an overflow of stack due to this platform size issues. Wilkinson, with the approach also using a verifier to support code security and boost execution speed, discloses passes to enable instructions each time they are fetched, to be rectified in view of the underflow or size discrepancies required by said platform requirements. Appellants fail to

provide convincing evidence that Yellin does prohibit any form of code modification in the purpose of improve runtime environment resources. Appellants fail to point how when Wilkinson approach by readjusting size is combined with Yellin code verifier method, the combined teaching would yield code deterioration, delayed runtime and would defeat the main purport of Yellin (or Wilkinson's) in such specifics whereby the combination would prove to teach away from Yellin's concern to timely handle conflicts due to code memory size, and stack overflows, in a heterogeneous platforms application. In response to Appellant's arguments against the references individually, one cannot show nonobviousness by attacking references individually where the rejections are based on combinations of references. See *In re Keller*, 642 F.2d 413, 208 USPQ 871 (CCPA 1981); *In re Merck & Co.,* 800 F.2d 1091, 231 USPQ 375 (Fed. Cir. 1986).

(H) Appellants have submitted that based on Yellin not teaching modifying of bytecode, the Office action rationale constitute impermissible use of hindsight, and that is more so, because on the negative effect for need to redevelop code for different platforms as evidence in col. 5 li. 14-31 of Yellin (Appl. Rmrks pg. 28-29). The portions proffered does not corroborate to Appellants' viewpoint that Yellin will adopt the bytecode verifier to prevent any further attempt to modify code; since Yellin only mentions that without the verifier, versions of code (received bytecodes in a target platform) during execution would require constant monitoring and otherwise would overburden the runtime efficiency. Wilkinson also teaches a verifier and does provide error handling with stop executing as in Yellin (see Wilkinson: Fig. 15); and Appellants mention that there would be no attempt to modify code in the fact that Yellin's verifier would block and stop further execution (Appl. Rmrks pg. 29, middle) cannot overcome the reasoning as

set forth in the rationale of the 103. One of ordinary skill in the art, learning of Yellin's use of a bytecode verifier and the same concern by Yellin and Wilkinson to address memory problems, would be motivated to support Yellin's contribution to solve platform issues (when versions of bytecodes are received from external sources) with the modifications by Wilkinson. In response to Appellant's arguments against the references individually, one cannot show nonobviousness by attacking references individually where the rejections are based on combinations of references. See *In re Keller*, 642 F.2d 413, 208 USPQ 871 (CCPA 1981); *In re Merck & Co.,* 800 F.2d 1091, 231 USPQ 375 (Fed. Cir. 1986).

(I) Appellants have submitted that the combination fails to render the invention obvious because the words of the claim have not been considered; and that, for example, four conditions of claim 11 must be met (Appl. Rmrks, pg 31). The language of the independent claims regarding 'converting' and 'changing ... to equal said second type ... is less than said second type' has been addressed at length from the above. The rationale addressing claim 11 falls under the ambit as to address compatibility of size of operand or instruction as these are to be fitted in a slot implemented by a register or runtime stack; and the rejection has provided why setting a type so that if it sensitive to overflow, would have been obvious in light of the combination of Yellin and Wilkinson (e.g. if a type is potentially larger than operand envisioned from a hardware stack, readjust the type to a larger type to fit its destined slot in the stack). The Appellants description of an arithmetic operation (Appl. Rmrks pg. 32) fails to convince how the obviousness as presented (re claim 11) would be deficient. As recited, the terminology regarding 'type' in the pertinent claim is not securely defined and consistent (refer to the above sections A, B, C) to enforce a entity to which 'type' is attributed to (type of instruction, operand, operator, return

value, said second type ?) or an entity necessarily specific to any arithmetic operation, notably, when the term like 'said second type' flows from the 'converting' step in claim 1, regarding which, Appellants fail to point out how by specific factual rebuttals, Yellin's reconfiguration cannot be complemented with Wilkinson's readjusting of bytecode to fit size in view of hardware constraints. Appellant's allegation that claim 11 requirements have not been met amounts to a lack of factual rebuttals showing how obviousness cannot be applied, in light of the overwhelming reuse of 'type' and lack of explicit specificity regarding the nature of 'type'

(J) Appellants have proffered a portion of the Specifications to distinguish what the Office Action construed as 'overflow' or 'underflow' in citing Yellin, and alleged (Appl. Rmrks pg. 33-34) that as such, *overflow* has nothing to do with stack storage availability as by Yellin. To that effect, Appellants point out that overflow is construed as underflow with a negative value from the Specifications, i.e. nothing related to stack; such that the Specifications has been ignored in interpreting the overflow. *Underflow* is understood as the opposite of overflow, and to represent overflow as a negative value of underflow (two negatives to mean one positive) would be a basic concept that would not require any extraneous inquiry beyond standard understanding of one of ordinary skill in that art. Overflow has been a well-understood concept, and it does not require further contribution from the invention for this concept to be understood and applied accordingly, and standard lexicographic meaning for it, based on the state of the art at the time of the invention, has been applied (i.e. when the target destination does not have sufficient storage for the intended content, overflow condition would occur). The Specifications (e.g. a negative value of underflow or arithmetic operation) cannot be read into the claim, nor can the language of claim 11 specific sufficiently to obviate 'overflow' in terms of overflow that would exclude

any stack implication, as used in Yellin. The argument is not persuasive to overcome the rationale (re claim 11) set forth in addressing what overflow entails and how obvious the setting of type would be in light of such potential threat. The Appellant's repeat about impermissible hindsight (Appl. Rmrks pg. 35) has been addressed earlier from above; that is, the endeavor of Yellin is deemed fit to be complemented with the code conversion as taught in Wilkinson, because both references, based on the teachings flowing their respective implemented approach, use of a verifier to expedite runtime, secure bytecodes from incurring runtime memory conflicts in addressing timely any potential overflows issue, a concern deemed common in both references.

In all, the claims stand rejected as set forth in the Office Action.

(11) Related Proceeding(s) Appendix

No decision rendered by a court or the Board is identified by the examiner in the Related Appeals and Interferences section of this examiner's answer.

For the above reasons, it is believed that the rejections should be sustained.

Respectfully submitted,

/Tuan A Vu /

Examiner Tuan Anh Vu, March 27 2009

Conferees:

/Lewis A. Bullock, Jr./
Supervisory Patent Examiner, Art Unit 2193

Eddie Lee:

/Eddie C. Lee/

Supervisory Patent Examiner, TC 2100